Zoea

Research note 37

A Grammar for Human Code

Edward McDaid & Sarah McDaid
10 Feb 2025

A programming language grammar describes the syntax of every valid program that can be written in that language. Yet the code produced by human developers corresponds to a minuscule and highly constrained subset of this ensemble. So how could we define a grammar that only describes human-like code?

The syntax of a programming language can be defined formally using a grammar. This identifies the atomic elements of the language such as variables, values, keywords and operators. It also specifies rules for how these elements can be combined to form statements, conditionals, control structures, functions and so on. Grammars of this kind are frequently included in programming language books and reference manuals. This is often the only direct exposure that most coders have to grammars.

A grammar also forms the core part of programming language compilers and interpreters. Here the grammar - itself encoded as software - is used to recognise the various language elements present in a piece of source code and determine how they are assembled. In this way, the compiler can ascertain whether the source code corresponds to a valid program where every element is recognised and all elements are combined according to some combination of the grammar rules. With invalid programs the compiler can indicate the nature of the problem and likely location of the error within the code.

A programming language grammar can also be useful for code generation since it describes all possible programs in that language. This means it is an easy matter to systematically generate every valid program in turn, starting with the smallest. However, the number of programs grows exponentially as program size increases. This means that

Copyright Zoea Ltd.

either, such generation is only feasible for small programs, or that some additional heuristic or filtering of the generated programs is required in order to reduce their number.

Generating code according to a grammar produces every possible program. This will include every program that might be produced by human coders. It will also include very many programs that - while they are syntactically correct - would never be written by a human programmer. This is because human code has a number of highly distinctive characteristics that differentiates it from other code. Currently, it isn't known whether human-like code is better, worse or no different to non-human-like code. In order to study human-like code we need a reliable way to generate it. First, we need to understand what makes human code different.

Human coders write code that corresponds to only a tiny fraction of all possible programs. This is manifest at the level of individual instructions where - rather than using every possible combination - people use a relatively small number of subsets. Similar skewing is also evident at the structural level. Furthermore, the fraction of human code versus all code decreases dramatically as program size increases. For an average sized function, only 1 in 10^50 (1 with 50 zeros) programs produced by syntax-driven code generation would resemble human code.

The reasons why this is the case are not well understood and are probably complex. A lot of it has to do with the fact that the frequency distribution of programming language instructions in human code is Zipfian. This means that a few instructions are used very frequently while most are rarely used - if at all. The use of words in human languages has a similar frequency distribution. It is thought that this may be required for effective communication or even to make language learning possible in the first place. Whatever the reason, the upshot is that programming language grammars are completely unsuitable for generating human-like code as most of the code they produce is non-human-like.

So how can we construct a grammar specifically for human code? One way might be to apply an existing grammar induction system to a very large sample of human code. Grammar induction is a process that can automatically turn a text sample into a grammar. A simple example is the Sequitur algorithm which can produce large grammars in linear time. One drawback with this approach is that the resulting grammar will only be capable of exactly reproducing the example programs that it was created from. In order to produce

Copyright Zoea Ltd.

human-like code that is similar to the sample code - but not identical - we also need to generalise the grammar. This would require a lot of effort but is certainly possible. However, the real problem with this approach is that generalisation would also cause the resulting grammar to produce an awful lot of non-human-like code. This is because any grammar produced that way would not capture the instruction cooccurrence patterns that exist in human code. Other possible approaches such as the use of probabilistic grammars would suffer from the same problem.

Zoea already uses these instruction cooccurrence patterns through what we call instruction subsets. These are simply sets of instructions that are used together in human code. Human developers do not use every combination of instructions but rather create software using a relatively small number of subsets of instructions. Instruction subsets used in Zoea encode these patterns. The process through which instruction subsets are created involves clustering and this leads to a significant level of generalisation without the wholesale ingression of non-human-like instruction combinations. Instruction subsets are currently used in Zoea to generate fragments of code and in doing so they can reduce the size of the search space by tens of orders of magnitude.

As it happens, it is a simple matter to transform an instruction subset into a grammar. Or rather, it is simple to use an instruction subset to specialise an existing language grammar. If we take the generic programming language grammar as a starting point, the first step is to remove all elements that are instructions but not members of the instruction subset. The next and final step is to remove any grammar rules, conjunctions and disjunctions that are now empty. The end result is a complete grammar that can only produce programs using that subset of instructions.

This subset grammar provides the same search space size reduction as instruction subsets. Furthermore, it is naturally generalised both because it retains the generalisation characteristics of instruction subsets and because it permits language elements including instructions to be used in any appropriate context. Being a grammar, it can generate complete programs rather than code fragments. Finally, a subset grammar is capable of generating much larger programs with given time and resources since it produces exponentially fewer programs than the original language grammar.

Copyright Zoea Ltd.

The above process results in a collection of subset grammars - one for each instruction subset. Each of these grammars produce different subsets of human-like programs. Taken together, the whole family of subset grammars covers the entire range of human-like software.

It is also possible to merge these grammars into a single grammar, if required. Merging a family of subset grammars is largely a matter of renaming non-terminal elements so that the rule sets of the component grammars can coexist within a single namespace. An extra disjunction is also required as the new root node - effectively to select between the rule hierarchies of the modified component grammars. The merged grammar retains all the benefits of the separate grammars and has a search space size that is simply the sum of its constituents. It may be possible to achieve a more elegant result using a 2-level Van Wijngaarden grammar but the basic principle would be the same.

So producing a grammar for human-like code turns out to be relatively straightforward. In particular, specialising an existing grammar as described is a remarkably simple yet useful technique.

Learn more at **zoea.co.uk**

Copyright Zoea Ltd.